

# Network-aware Mobile Programs\*

M.Ranganathan, Anurag Acharya, Shamik D. Sharma and Joel Saltz  
Department of Computer Science  
University of Maryland  
College Park, MD 20740

## Abstract

*In this paper, we investigate network-aware mobile programs, programs that can use mobility as a tool to adapt to variations in network characteristics. We present infrastructural support for mobility and network monitoring and show how adaptalk, a Java-based mobile Internet chat application can take advantage of this support to dynamically place the chat server so as to minimize response time. Our conclusion was that on-line network monitoring and adaptive placement of shared data-structures can significantly improve performance of distributed applications on the Internet.*

## 1 Introduction

A mobile program can move code, data or an active thread of control from site to site in a distributed environment. This flexibility has many potential advantages. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-site instead of fetching all the data to its current location. Similarly, an internet video-conferencing application can minimize overall response time by positioning its server based on the location of its users. The primary advantage of mobility in these scenarios is that it can be used as a tool to adapt to variations in the operating environment. Applications can use online information about their operating environment and knowledge of their own resource requirements to make judicious decisions about placement of code, data and threads of control.

In this paper, we investigate *network-aware* mobile programs, i.e. programs that can position their computational elements and data structures based on their knowledge of network characteristics so as to improve their performance. In particular we investigate what policies are suitable for making mobility decision? Second, is the variation in

network characteristics such that adapting to them proves profitable. Finally, can adequate network information be provided to mobile applications at an acceptable cost.

In order to adapt to network variations, mobile programs must be able to decide when to move, what to move and where to move. There are three types of network variations which may be cause for migration: (1) population variations, which represent changes in the distribution of users on the network, as sites join or leave an ongoing distributed computation; (2) spatial variations, i.e. stable differences between in the quality of different links, which are primarily due the host's connectivity to the internet; and (3) temporal variations, i.e. changes in the quality of a link over a period of time, which are presumably caused by changes in cross-traffic patterns and end-point loads. Spatial variations can be handled by a *one-time placement* based on the information available at the beginning of a run. Adapting to temporal and population variations requires *dynamic placement* which needs a periodic cost-benefit analysis of current and alternative placements of computation and objects. Dynamic placement decisions have two partially conflicting goals: maximize the performance improvement from mobility and minimize the cost of mobility. If an opportunity for improving performance presents itself, it should be capitalized upon; however, reacting too rapidly to changes in the network characteristics can lead to performance degradation as the performance gain may not offset the mobility cost.

We investigate these issues in the context of *Sumatra*, an extension of the *Java*<sup>1</sup> programming environment [10] that provides a flexible substrate for adaptive mobile programs. Since, mobile programs are scarce, we developed our own mobile internet chat server. This application, called *adaptalk*, monitors the latencies between all participants and locates the chat server so as to minimize the maximum response time. We selected this application since it is highly interactive and requires fine-grain communication. If such an application is able to take advantage of information about network characteristics, we expect that

---

\* This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

---

<sup>1</sup>Java is a registered trademark of Sun Microsystems

many other distributed applications over the internet would be similarly successful. The resource that governs the migration decisions of *adaptalk* is network latency. To provide latency information, we have developed *Komodo*, a distributed network latency monitor.

To evaluate if mobile applications can take advantage of network-awareness, we examined the performance of *adaptalk* with and without mobility. Our evaluation had two main goals: (1) to determine the performance benefits, if any, of network-aware placement of the central chat server over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Our results are encouraging - they indicate that on-line monitoring and dynamic placement can significantly improve performance of distributed applications on the Internet.

This paper is a first step in demonstrating that distributed programs can use mobility as a tool to adapt to variations in their operating environment. The main contribution of this work is that it shows the feasibility and profitability of this approach. We establish feasibility by providing a programming interface and efficient system support for thread and object migration in Sumatra. Our experiments with *Komodo* and *adaptalk* indicate that network-aware programs can successfully use mobility to adapt to spatial and temporal variations in network latency over the internet.

The paper is organized as follows. Section 2 describes Sumatra and the programming model that it provides. Section 3 describes the design and implementation of *Komodo*. Section 4 describes the *adaptalk* application and the policy it uses to make mobility decisions. Section 5 describes our experiments and presents the results. Section 6 discusses the results and their implications. Section 7 describes related work and Section 8 provides our conclusions and plans for future work.

## 2 Sumatra: a Java that walks

Sumatra is an extension of the Java programming environment that supports adaptive mobile programs. Platform-independence was the primary rationale for choosing Java as the base for our effort. In the design of Sumatra, we have not altered the Java language. Sumatra can run all legal Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter without affecting the virtual machine interface.

Our design philosophy for Sumatra was to provide the mechanisms to build adaptive mobile programs. Policy decisions concerning when, where and what move are left to

the application. The main feature that distinguishes Sumatra from previous systems [3, 6, 24, 13] that support mobile programs is that *all* communication and migration happens under application control. Furthermore, combination of distributed objects and thread migration allows applications the flexibility to dynamically choose between moving either data or computation. The high degree of application control allows us to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources. We believe that the space of design choices for adaptive mobile programs is yet to be mapped out and such flexibility is important to help explore this space.

Sumatra adds two programming abstractions to Java: *object-groups* and *execution engines*. An object-group is a dynamically created group of objects. Objects can be added to or removed from existing object-groups. All objects within an object-group are treated as a unit for mobility-related operations. This allows the programmer to customize the granularity of movement and to amortize the cost of moving and tracking individual objects. This is particularly important in languages like Java because every data structure is an object and moving the state, one object at a time, can be prohibitively expensive. An *execution-engine* is the abstraction of a location in a distributed environment. In concrete terms, it corresponds to an interpreter executing on a host. Sumatra allows object-groups to be moved between execution-engines. An execution-engine may also host an active thread of control. Currently, the implementation does not support multi-threaded mobile programs. Threads can move between execution-engines.

The principal new operations provided by Sumatra are:

**Object Migration:** Objects on the heap can be checked into or out of an object group at application request. Object groups may be moved between engines. During motion, objects in the object group are automatically marshalled using type-information stored in their class templates. When an object-group is moved, all local references to objects included in the group (stack references and references from other objects) are converted into *proxy references* where the new location of the object is recorded. Some objects, such as I/O objects, are tightly bound to local resources and cannot be moved. References to such objects are reset and must be reinitialized at the new site. The class template (and associated bytecode) for an object can be downloaded into an execution-engine on application request. Downloaded class-templates are cached; the `ClassLoader` checks this cache before checking the local file system.

**Remote Method Invocation:** Method invocations on proxy objects are transparently translated into calls at the remote site. Type information stored in class-templates is used to achieve RPC functionality without a stub compiler.

Exceptions, generated at the called site are forwarded to the caller. If an object is shared between threads on different engines, it is possible that the object can move without the knowledge of one or more engines. In such cases, sending a remote method invocation to the expected site of the object returns an *object-moved* exception along with a new forwarding address to caller. The caller can handle the exception as it deems fit (e.g., re-issue the request, migrate to the forwarded location, raise a further exception and so on).

**Thread migration:** Sumatra allows explicit thread migration using a `engine.go()` function that bundles up the stack and the program counter and restarts execution at the specified execution-engine. To automatically marshal the stack, the Sumatra interpreter maintains a type stack, parallel to the value stack, which keeps track of the types of all variables on the stack. When a thread migrates, Sumatra transports with it, all objects that are referenced by the stack but are not a part of any object-group. All stack references to objects that are left behind (i.e. were part of some object-group) are converted to references to proxy objects. After migration, many of the proxy references on the stack may actually refer to objects that are on the destination site; these references are converted to local references before the call to `go` returns.

**Remote execution:** A new thread of control can be created by *rexec*'ing the `main` method of a class existing on a remote engine. The arguments for the invocation are copied and moved to the remote site. Unlike remote method invocation, remote execution is non-blocking; the calling thread resumes immediately after the `main` method call is sent to the remote engine. Currently, Sumatra imposes the restriction that concurrent threads must execute on different engines. Concurrent threads communicate using calls to shared objects. The thread initiating a remote execution can share objects with the new thread by passing it references to these objects as arguments to `main`.

Thread migration allows movement of control at arbitrary points in the execution. For example, a remote method invocation may migrate based on some data or environment dependent condition - thereby causing the control to return from a location different from the original target of the remote invocation.

**Remote Signal Handlers:** Sumatra allows the user to write a signal handler in Java. This allows remote signal handlers to be installed. Signal handlers are a simple method of implementing "up-calls". For example, a server that produces periodic data may place the data in some "well known" location and signal the engine in which a signal handler is installed. The signal handler may then pick up the data from the well known location.

## 2.1 Example

Say we want to look through a database of X-ray images stored at a remote site and apply a quick selection algorithm to extract "interesting" lung images. These images are then subjected to a more compute-intensive cancer-detection process. One way to write the program would be to download all images from the image server and do all the processing locally. This may cause long delays due to the network traffic involved. Another approach would be to send the selection procedure to the site of the image database. Only "interesting" images would be sent back to the main program, greatly reducing network requirements. A third, and even more flexible approach would allow the shipped selection procedure to extract all the interesting images from the database but return only the *size* of the extracted images to the main program. If the size is still too big, the program may choose to move itself to the database site and perform the cancer-detection computation there rather than downloading all the data - thereby avoiding the network bottleneck while paying the cost of slower processing at the server. On the other hand if the size is small, the data can be shipped over and processed locally with a faster native method. The code in Figure 1 shows this adaptive version of the code. This program makes its decision to migrate in a rudimentary fashion; a more realistic version of this application would also take network bandwidth and the relative processing power available on both machines into account for migration decisions.

We next describe a distributed monitor that allows network latency information to be gathered by Sumatra applications in order to make mobility decisions.

## 3 Komodo: a distributed network latency monitor

Komodo<sup>2</sup> is a distributed network latency monitor. The design principles of Komodo are: low-cost active monitoring and fault-tolerance. Active monitoring uses separate messages for monitoring, passive monitoring generates no new messages and piggybacks monitoring information on existing messages. An active monitoring approach is needed for *adaptalk* (described in the next section) as passive monitoring cannot provide information about links that are not used in the current placement but could be used in alternative placements. It is our working hypothesis that effective mobility decisions can be based on medium-term (30sec-few minutes) and long-term (hours) variations. At these resolutions, we believe that active monitoring can be

---

<sup>2</sup>Komodo dragons are a species of *monitor* lizards found on the island of Komodo which is close to both Java and Sumatra.

---

```

.....
lung_object = new Lung();
myengine = System.rpc.myEngine();

// Create an engine at the xray database site.
remote_engine = new Engine("xrays.gov");
// Send the lung class-template to the remote engine
remote_engine.downloadClass("Lung");
// Create a new object group.
objgroup = new ObjGroup("lung_group");
// Add the lung_object to the object group
objgroup.checkIn(lung_object);
// Move the object group to the database site
objgroup.moveTo(remote_engine);

// a remote method call selects interesting xrays
size = lung_object.query(db, "BigLungs");

// Are there too many images to bring over?
if ( size > too_many_images ) {
    // Migrate thread, process images and return.
    remote_engine.go();
    result = lung_object.detect_cancer();
    myengine.go();
}
else {
    // there are only a few interesting xrays. Fetch them
    // and process locally, using a faster native method.
    objgroup.moveTo(myengine);
    result = lung_object.n_detect_cancer();
}

// display result locally
System.display(result);

```

---

Figure 1: Excerpt of a Sumatra program that adaptively migrates to reduce its network bandwidth requirements

---

achieved at an acceptable cost. This section briefly describes the design and implementation of Komodo.

Komodo allows applications to initiate monitoring of network latency between any pair of hosts running the monitor. The application need not be resident on one of the hosts in the host-pair being monitored. Komodo is implemented as a user-level daemon that runs on every host. Applications pass monitoring requests to their local Komodo daemon. If the requested link includes the current host, the local daemon handles the request. Otherwise, it forwards the request to the daemon on the appropriate host. Each daemon monitors the UDP-level latency on a network link for which it has received monitoring requests, by sending a 32-byte UDP packet to the daemon on the other end of the link of interest. If an echo is not received within an expected interval, (the maximum of the ping period or five times the current round trip time estimate) the packet is retransmitted. Using UDP for communication may, occasionally, lead to loss of messages. Message loss can lead only to a short-term loss of efficiency. As we expect monitoring requirements to be coarse-grained, the effect of packet loss should be small.

Applications that initiate a monitoring request can control the frequency with which Komodo *pings* the specified link to a maximum upper bound. Applications need to refresh requests periodically to keep them alive; otherwise, Komodo drops the request after a specified period of time.

The latency measures acquired by Komodo are passed through a filter before being provided to the applications. This filter eliminates singleton impulses as well as noise within a jitter threshold (we use a jitter threshold of 10ms, which is the resolution of most Unix timers). If the measure

changes rapidly, a moving window average is generated. The filter produces a stepwise constant output which reflects the average trend of the ping measurements. This filter was designed on the basis of our study of a large number of internet latency traces (see Section 5.1) which revealed that: (1) there is a lot of short-term jitter in the latency measures but in most cases, the jitter is small; (2) there are occasional jumps in latency that appear only for a single ping; (3) for some traces, the latency measure fluctuates rapidly; (4) the rtt measures show an "approximately stepwise" behavior - that is a definite trend (median value within a window) which has abrupt jumps. Similar observations have been made by others [5]. Figure 2(a) illustrates the operation of the filter.

Each daemon maintains a cache of current latency estimates for all its currently active monitoring requests. This cache is maintained in a well-known shared memory segment and can be efficiently read by all Sumatra applications executing on the same machine. Cooperating Komodo daemons forward latency information in response to persistent remote requests. A latency estimate for a request received from another host is forwarded only when a new filtered estimate (different from the previous filtered estimate) is generated and is piggy-backed onto a ping reply if possible.

To address concerns about the cost of active monitoring, we measured the CPU utilization of Komodo for varying number of links. We determined that the maximum CPU utilization for up to eight links is less than 0.4 %. The amount of data transferred is 256 bytes per second. Also, for up to eight links, the CPU utilization scales linearly. By extrapolation, the load for 20 hosts would be 1% CPU

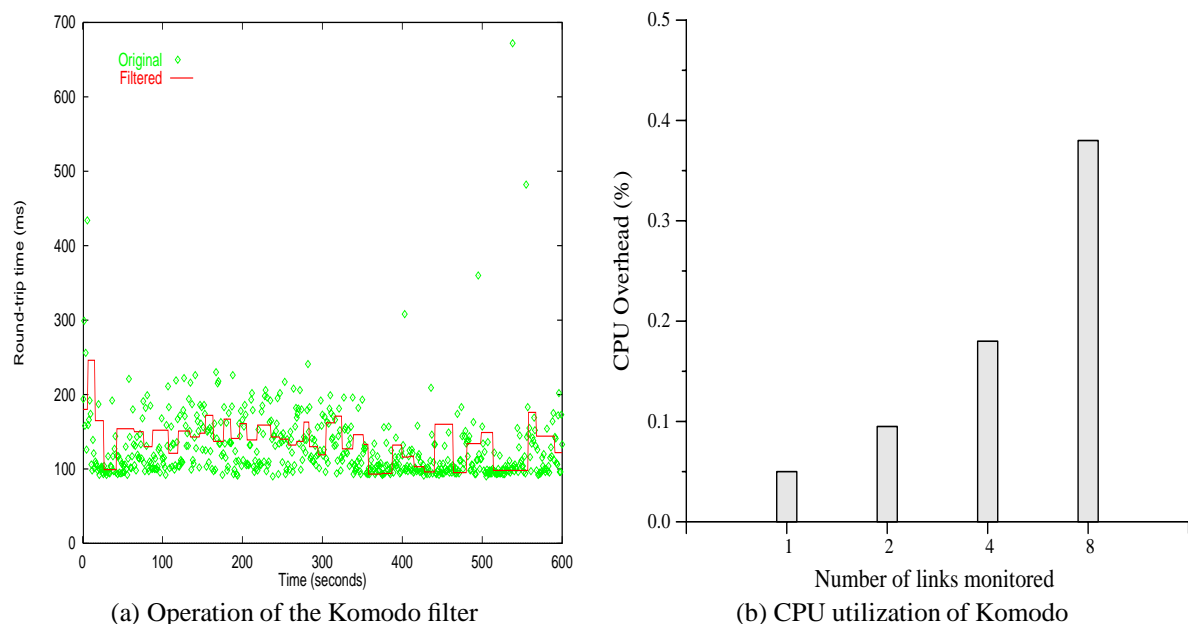


Figure 2: (a) The input to the filter is a 10-minute trace of one-per-second latency measures between `baekdoo.cs.umd.edu` and `lanl.gov`. Note that the four single-ping impulses towards the right end have been eliminated. (b) The CPU utilization is computed by dividing the (user+system) time by the total running time. Each experiment was run for 1000 seconds with one ping per second for all links.

utilization and 640 bytes/sec data transfer.<sup>3</sup> Further details about Komodo are presented in [23].

## 4 Adaptalk: An adaptive internet chat application

Adaptalk is a relatively simple network chat application built using Sumatra and Komodo. It allows multiple users to have an online conversation; new participants can join an ongoing conversation at any point; multiple independent conversations can be held. To ensure that all participants see the same conversation and that new participants can join ongoing conversations, a central server is used to serialize and broadcast the contributions.

Adaptalk has been divided into three components: handling keyboard events, managing the chat screen and coordinating the communication between participants. Each component is implemented by a separate *object-group*. Each host participating in the conversation runs two *execution-engines*, one houses the *screen* object-group and the other houses the *keyboard* object-group. The central server is implemented as a separate shared object-group, *msgboard*, which can (and does) move between

hosts participating in the conversation, positioning itself within the engine that houses the screen on its current host. Each message issued by a participant starts from a keyboard which invokes a remote method on the *msgboard*. The *msgboard* serializes incoming messages and issues a succession of *remote-execution* requests, one per participant, that updates the screens on all participants. In this case, remote execution is preferred to remote method invocation as there is no useful return value and remote execution allows fast one-way communication.

Individual messages in adaptalk, and other chat applications, consist of single lines of characters, usually no more than 50-60 characters. The goal of a chat application is to provide a short response-time to all participants so that a conversation can make quick progress. The response-time for a particular participant depends on the latency between it and the central server. Given the latencies of all the links, the primary knob that adaptalk can turn, to maintain a low response-time for all participants, is the position of the central server.

### 4.1 Mobility policy

There are two main features of the adaptalk mobility policy. (1) continuous tracking of the instantaneously *most-suitable-site* and (2) deferral of server-motion till the potential for a significant and *stable* performance advan-

<sup>3</sup>This assumes that the linear scaling holds; given the low utilization there is no reason to believe that it would not.

tage has been seen. The first feature allows it to quickly take advantage of opportunities for optimization; the second helps ensure the gain is greater than the cost. The goal of *adaptalk* is to minimize the maximum response-time seen by any participant. The migration policy of *adaptalk* tries to minimize this cost function.

As mentioned in the previous section, the goal of *adaptalk* is to minimize the maximum response-time seen by any participant. The suitability of a machine as the location of central server is characterized by the maximum over the network latency measures for all participants. The machine that achieves the lowest measure is designated the *most-suitable-site*.

We expect three types of variations in the network characteristics which may be cause for migration: (1) population variations, which represent changes in the distribution of users on the network, as participants join or leave an ongoing conversation and their machines become available (or unavailable) as potential locations to place *msgboard*; (2) spatial variations, i.e. stable differences between latencies of different links; and (3) temporal variations, i.e. changes in the latency of a link over a period of time.

*Adaptalk*'s migration policy, shown in pseudo-code in Figure 3, can adapt to all three types of variations. This algorithm is run at the location that hosts the *msgboard* each time a message is posted. The algorithm runs as part of the *post\_msg* method at the *msgboard*.

A new site for the central server is selected whenever: (1) one of the sites that does not currently host the server receives more than a threshold score; or (2) the current site receives a very low score over a threshold number. The first condition is used to move the server to locations that consistently promise better performance; the second condition is used to quickly move away from locations that provide poor performance.

A count is maintained for each host where *msgboard* may be housed. This count is initialized to 0 and is cleared when a participant enters or leaves the conversation. Consider the case with a fixed number of participants with significant spatial variation in network latency and little temporal variation. In this case, the migration algorithm rapidly recognizes the best location for the *msgboard*, but waits until this choice has been ratified over some period of time ( $\text{count} > \text{win\_threshold}$ ) before moving *msgboard*. As shown in Section 5, this policy allows *adaptalk* to effectively insure itself against poor *initial placement* of the *msgboard*. Once a good location has been found, the *msgboard* does not move, unless temporal variations or changes in population distribution cause another node to become a substantially better location (i.e.  $\text{count}[w] > \text{win\_threshold}$ ) or

the current host to become a substantially bad choice (i.e.  $\text{count}[\text{curr\_engine}] < \text{loss\_threshold}$ ). In such cases, the *msgboard* will move during the conversation. After initial experiments with *adaptalk*, we set the *win\_threshold* to be  $25 \times n$ , the *loss\_threshold* to be  $12 \times n$  and the *decision\_cycle* to be  $50 \times n$ . Here,  $n$  is the number of participants. The length of the *decision\_cycle* was set large enough to amortize the cost of movement in cases where large temporal variations or fluctuations in population distribution cause frequent repositioning.

---

```

.....
Get the all to all latency map from Komodo;
Find the site s that would minimize the max
    latency for messages posted to msgboard;
count[s] = count[s] + 1; rounds++;
let w be the site with the largest count;
let curr_engine be the engine which
    currently houses msgboard;
// Found a clear cut winner.
if (count[w] > win_threshold) return w;
else if (rounds % decision_cycle == 0) {
    // Is the current engine a bad loser ?
    if (count[curr_engine] > loss_threshold) {
        clear count for each host;
        return curr_engine;
    } else {
        // Current engine is a bad loser.
        let new_host be the host with the
            maximum count;
        clear count for each host;
        return new_host;
    }
} else return null; // cycle not yet over.

```

---

Figure 3: Decision Algorithm for *msgboard* placement used in *Adaptalk*. This algorithm is run at the location where the *msgboard* resides each time a message is posted.

## 5 Evaluation

To evaluate the performance impact of network-aware adaptation on the Internet, we performed two sets of experiments. First, we monitored round-trip times for 32-byte ICMP packets sent to a large set of hosts over several days. The goal of these experiments was to study the spatial and temporal variation in network latency on the Internet. Results from this study are presented in section 5.1.

Second, we measured the performance of three versions of *adaptalk* over long-haul networks, using traces collected during the internet study. Our evaluation had two main goals: (1) to determine if network-aware placement

of components of an application distributed over multiple hosts on the Internet provides significant performance gains over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Results from this study are presented in section 5.3.

## 5.1 Variations in Internet latency

We selected 45 hosts: 15 popular .com web-sites (US), 15 popular .edu web sites (US) and 15 well-known hosts around the world. These host were pinged from four different locations in the US. The study was conducted over several weekdays, each host-pair being monitored for at least 48 hours. We used the commonly available ping program and sent one ping per second. This resolution was acceptable as our goal was to discover medium-term (30sec/minutes) and long-term (hours) variations.

The conclusions of our study, briefly, are: (1) there is large spatial variation in internet latency (the per-hour mean latency varied between 15 ms and 863 ms for US hosts and between 84 ms and 4000 ms for non-US hosts); (2) there is a large and stable variation in the latency of a single host-pair over the period of a day (maximum daily variation in per-hour mean latency for US hosts was 550 ms and for non-US hosts was 5750 ms); (3) There is a lot of jitter in the latency measures but in most cases, the jitter is small. (4) There are isolated peaks in latency that appear only for a single time interval.

## 5.2 Experimental Setup

Having established that there are significant spatial and temporal variations in network latency on the internet, we examined how well *adaptalk* could adapt to these variations.

To simulate the characteristics of long-haul networks, we decided to run our experiments over a low-latency LAN and delay all packets based on the ICMP ping traces described above (see Figure 4 (a)). This approach also allowed us to perform repeatable experiments. To ensure that delaying packets, instead of using a real network, does not skew the latency measures, we performed a simple test. Free-running Komodo monitors were installed at `bookworm.cs.umd.edu` and `jarlsberg.cs.wisc.edu` and were used to collect UDP latency measures between this host-pair. In parallel, a trace of ICMP ping times between these two hosts over the same period (5000 sec) was collected. This trace was later fed into trace-driven Komodo monitors running on two hosts on our LAN. The latency measures reported by the trace-driven monitors matched quite well with the

actual latency measures reported by free-running monitors. The average of the actual latency measures was 128 ms (std dev = 64); the average of the values reported by the trace-driven monitors was 144 ms (std dev = 68).

We performed all our experiments on four Solaris machines on our LAN. We picked six trace-segments from the internet study and used them to delay packets between the machines. All these segments were over the noon-2pm EDT period.<sup>4</sup> These traces were selected to approximate the network latency spectrum observed in the internet study. Hosts participating in the selected traces include: `java.sun.com`, `home.netscape.com`, `www.opentext.com`, `cesdis.gsfc.nasa.gov`, `www.monash.edu.au` and `www.ac.il`. This setup makes the four local machines behave like four far-flung machines on the internet. Figure 4 (b) shows the configuration used for the experiments.

## 5.3 Experiments

We performed a series of experiments to evaluate the benefits of adapting to population variation, spatial variation and temporal variation. The experiments consisted of running three different versions of the chat server. The first version, called *static*, had no migration support and no network-awareness. The location of the `msgboard` was chosen in a network-oblivious fashion. The second version was a stripped-down version of *adaptalk*, called *one-shot*. It used network information from Komodo to find the best initial placement for the `msgboard`, and used mobility support to move it there. After initial placement, migration decisions and network-awareness were turned off. The third version, called *dynamic* was the full-fledged *adaptalk*, as described in section 4. It used on-line monitoring and dynamic placement to position the `msgboard`.

The performance of *static* depends on the location of the `msgboard`. If *static* chooses the same location as *one-shot*, both would have the same performance. On the other hand, since *static* is network-oblivious, it is just as likely to place the `msgboard` at the worst possible location. As the performance of *one-shot* already presents a rough upper-bound of *static*'s performance, we deliberately chose the worst initial placement when running *static*.

**Adaptation to Population Variation:** To evaluate the effect of changing user distribution we used the following workload: A conversation was initiated between hosts C and D. Host B joins the conversation after 15 minutes, and host A joins after another 15 minutes. Each host sends a sequence of 70-character sentences with a 5-second think time between sentences. With only two hosts initiating the

<sup>4</sup>12 noon is the beginning of the daily latency peak for US networks and the end of the daily latency peak for many non-US networks.

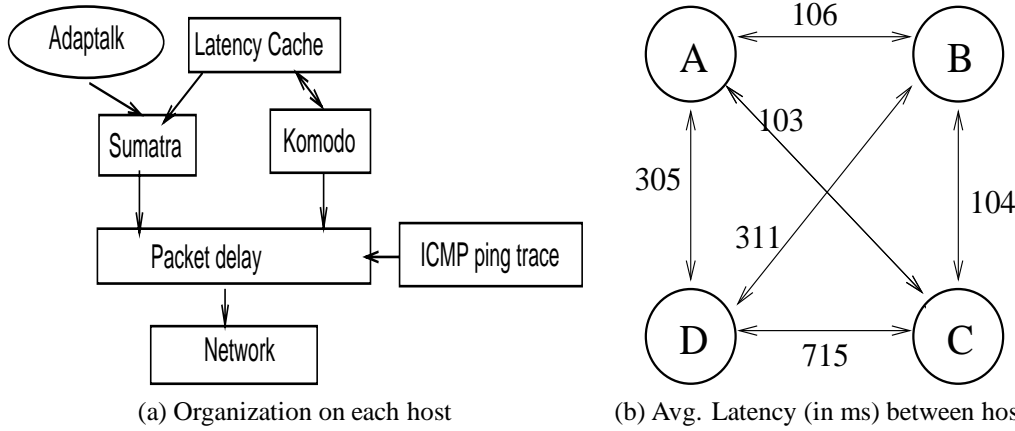


Figure 4: Experimental Setup. Four local machines on a LAN were used to simulate four remote machines on the Internet by adding delays to packets. ICMP ping traces between real Internet hosts were used to generate the delays, so as to capture real-life temporal variations in latencies.

conversation, there is no difference between the best and worst initial placement for the `msgboard` and both the *static* and *one-shot* versions perform identically (both place the `msgboard` on host D). Figure 5 (a) plots the maximum latency over all hosts for the *one-shot* version. Note that even after new hosts join the conversation there is no noticeable difference in maximum latency. In contrast, the *dynamic* version adapts to the changing population workload. Soon after host B joins the conversation, the adaptive placement policy moves the `msgboard` there, causing a drop in the maximum latency. After host A joins the conversation, the `msgboard` moves between hosts A and B in response to temporal fluctuations. This can be seen from the variation in latency for host B in Figure 5 (b). These movements help keep the maximum latency steady even in the presence of temporal fluctuations.

**Adaptation to Temporal and Spatial Variation:** In this case the client population is assumed to be stable. The workload consists of all 4 hosts jointly initiating a conversation which runs for 75 minutes. As before, each host generates a new sentence every 5 seconds. In this case, the network-oblivious (*static*) version places the chat server on host D. The network-aware (*one-shot*) version uses latency information provided by Komodo to determine that host B is a much better placement. For the *dynamic* version, initial placement is less important as it should be able to recover from a bad initial placement. For this version, we place the `msgboard` at host D, the worst-possible location.

To avoid clutter, Figure 6 shows the performance of these three versions in two different graphs. Figure 6 (a) compares the maximum latency (over all participants) for the *dynamic* and *static* versions. As seen from the sharp drop on the left end of the graph, the *dynamic* version is

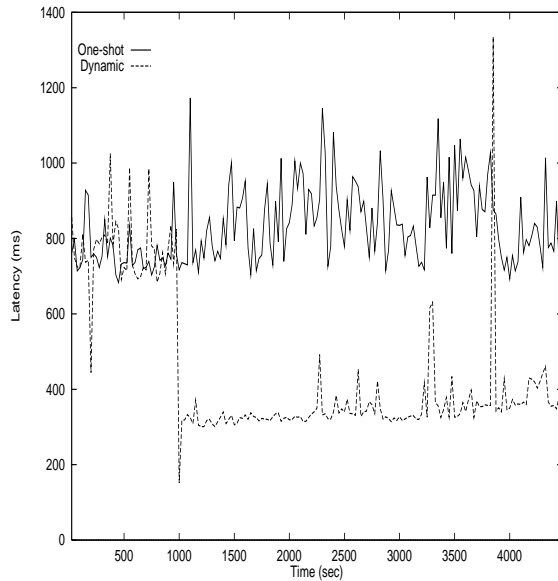
successfully able to move the `msgboard` away from its bad initial placement to more suitable location. Figure 6 (b) compares the average maximum latency (over all participants) for the *dynamic* and *one-shot* versions. It shows that once the *dynamic* version moves the server to a more suitable location, the performance of the two versions is largely equivalent. This implies that adapting to short-term temporal variations in a steady population workload does not provide much performance advantage over one-shot network-aware placement. It may, however, still be advantageous to adapt to long-term temporal variations. We note that at the far right of graph Figure 6 (b), temporal variation in the link latencies do allow the *dynamic* version to do better than the *one-shot* version.

## 6 Discussion

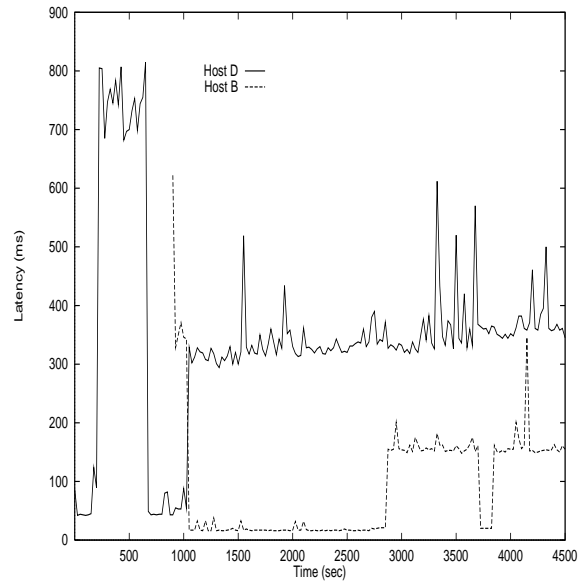
In the introduction, we had raised three questions with respect to network-aware mobility. First, how should programs be structured to utilize mobility to adapt to variations in network characteristics? Second, is the variation in network characteristics such that adapting to them proves profitable? Finally, can adequate network information be provided to mobile applications at an acceptable cost?

Our experience with Sumatra and `adaptalk` provides some early insights about application structure suitable for adaptive mobile programs. First, the migration policy should be cheap so that applications don't have to analyze the tradeoffs of the migration decision itself. An easy-to-compute policy allows frequent decisions and rapid adaptation to changes in the environment. We believe that an easy-to-compute migration policy was key to `adaptalk`'s ability to quickly find good locations for the chat server.





(a) one-shot vs. dynamic for population variation.  
Max latency (over all participants) vs time.



(b) Latency variations for hosts B and D  
Jumps signify movement of the server.

Figure 5: Adaptation to Population variation. Hosts C and D initiate the conversation. Host B joins after 900 seconds and host A joins after 1800 seconds. The *one-shot* version places the chat server at host D. The *dynamic* version migrates the server when new hosts join.

Second, good modularization helps an application take advantage of mobility. Modularization is important for all distributed applications but it is more so for mobile programs as they have to make online decisions about the placements of different components. An important question that needs further investigation is where to place the control for the mobility decisions (ie. should the logic for the mobility decisions be placed along with the object that moves or is central control adequate.)

To answer the second question, we evaluated the profitability of adapting to changes in the user-distribution as well as spatial and temporal variations in network latency. Adapting to changes in user-distribution led to significant gains allowing *adaptalk* to find better placements as more users came online. Support for mobility allows applications built around a central data-structure to recover from a poor initial placement of this structure by repositioning it to a more suitable location. Adapting to temporal variations alone did not lead to significant benefits over the period of an hour and a half. In light of this experience, we expect that a simpler migration policy for *adaptalk* for short periods would consider migration only when users join or leave the conversation, rather than on every message as is currently done. Since long-term variation of latency could be as large as 550 ms (US hosts) and 5750 ms (non-US-hosts), longer conversations could still benefit from adapting to

temporal variations.

Our experiments with Komodo illustrate that cheap active monitoring can provide network information that can be profitably exploited. Though it would be best to use Komodo as a stand-alone system supplying network information to many distributed applications, its cost is so low that one can contemplate rolling Komodo into individual applications such as *adaptalk* without overloading the network. Active monitoring was needed for *adaptalk* as it needed information about links that are not used in the current placement but could be used in alternative placements. Active monitoring, as implemented in Komodo, will not be as cheap for applications that are bandwidth-sensitive and not latency-sensitive. We are currently investigating methods to cheaply estimate Internet bandwidth.

Finally, we would like to argue the need for mobility as an adaptation mechanism. An alternative adaptation mechanism, which places replicated servers at all suitable points in the network, could adapt to spatial, temporal and population variation by handing off conversations between servers and by using dynamically created hierarchies of servers. It is quite likely that for any particular application, such a strategy would be able to achieve the performance achieved by programs that use program mobility as the adaptation tool. The advantage of mobility-based strategies is that it allows small groups of users to rapidly set up

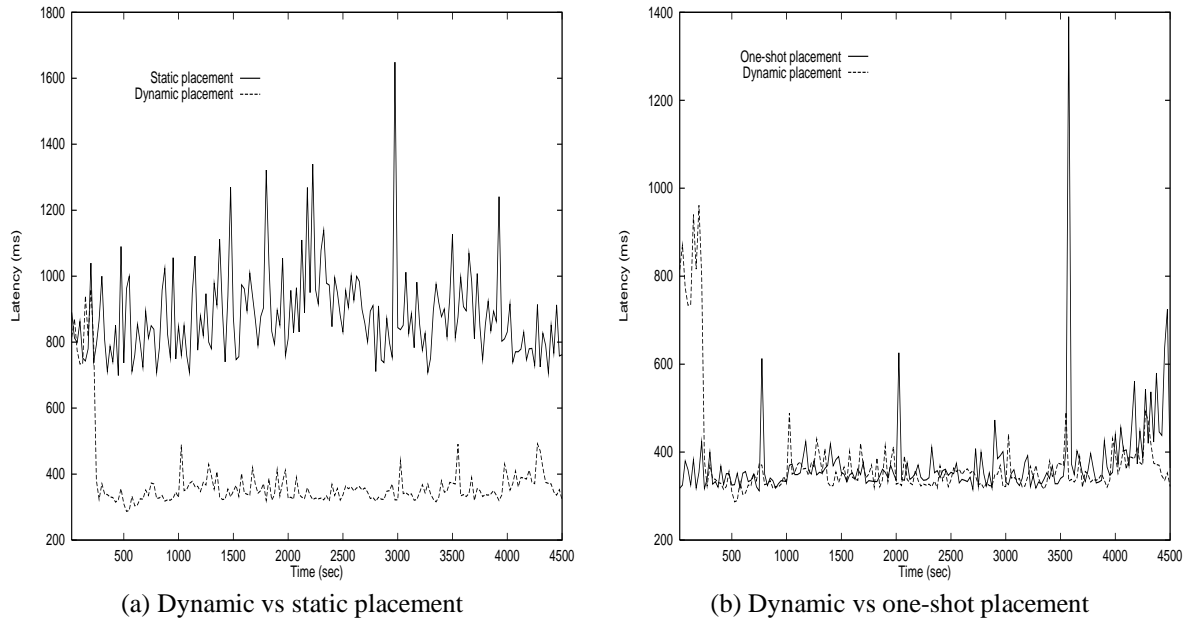


Figure 6: Maximum latency (over all participants) vs time in adaptalk. The one-shot and static worst placement are computed based on latency information available when the conversation is initiated. The population of talkers is constant.

private communities on-demand without requiring extensive server placement. This is facilitated by the fact that mobility-based strategies can automatically determine and utilize suitable server locations.

## 7 Related work

The idea of moving code, objects and processes around a distributed system to achieve better performance and utilization is not new. There are several distinct classes of systems that support mobility. A rough classification may be (1) systems that support remote evaluation; (2) systems that support passive object migration; (3) systems that support active thread migration; and (4) systems that support a combination of these features.

Remote evaluation (or code shipping) has been supported by several systems - for example Java [10], REV [17], REXDC [4] NCL [15] and the UNIX *rsh* facility and Avalon/Common Lisp [25].

Movement of passive objects (ie. data) is supported by Java Object Serialization [1] and Modula 3 Network Objects [2].

Several systems have been built which permit an executing program to move while it is in execution<sup>5</sup> - for example *Obliq* [3], Agent TCL [24], *Emerald* [16], Telescript [12]

<sup>5</sup>These systems are also called *Mobile Agents* or *Itinerant Agents* in the literature

and TACOMA [14]. We examine some of these systems below.

Process migration is similar to mobile programs. The difference is that process migration moves the entire execution image of a process in execution. This complicates matters - especially for I/O objects [8] and kernel state. Process migration has been used in homogeneous networks of workstations to provide better performance, utilization and load distribution [7, 8, 20, 26]

Several distributed object systems that support mobility have been built. Our system design has been most influenced by Emerald, Obliq and Telescript which we describe briefly below.

Emerald [16] is an object-based language and system designed for the construction of distributed programs. Emerald supports a "pure" object oriented model. An explicit goal of Emerald is support for object mobility. Objects in Emerald can freely move within the system to take advantage of distribution and dynamically changing environments. Emerald objects can be active (ie. having an associated process) or passive. Emerald provides fast access to local objects by avoiding indirection. Such objects are immobile. Mobile objects are accessed through one extra level of indirection and hence are slower to access. These are known as *Global* objects. In contrast all objects are "*Global*" objects in Sumatra. Emerald supports object groups and different models of passing parameters to remote method invocations. Emerald runs on a homogeneous network of workstations.

Obliq [3] is an object-oriented system that is based on Modula-3 network objects [2]. Obliq supports enforces adherence to static scoping in a distributed environment and the provides transmission of closures and objects. Objects in Obliq don't move from a site. However, the state of an object can be cloned and transmitted to another site.

Telescript [12] is a commercially available agent system. It supports the notion of *agents*, *places* and *Engines*. An *agent* is a mobile program. A *place* is a physical location. An *Engine* is an interpreter in an infinite loop (as in Sumatra). Agents migrate from *engine* to *engine* by using a "go" instruction. Telescript provides an *authority* based identification and authentication model. An *authority* defines the identity of the agent. Agents and *places* may discern but not falsify authorities. Agents are allocated permits when they arrive at a *place*. A permit defines what the *agent* may do at the *place*. It may define restrictions on resource usage, whether the agent may create other agents etc. An agent that exceeds its allocated *allowance* of resources is destroyed. An agent may impose temporary restrictions on its own resource usage. The system then notifies it when these temporary restrictions are reached. Agents may *meet* at pre-arranged meeting *places* and exchange information. Telescript does not support remote references. We have not as yet addressed the problems of resource containment and security in Sumatra.

The TACOMA system [14] defines an execution model where agents can migrate from site to site and rendezvous with each other. Data is packaged and placed into *folders* which agents carry from site to site. Agents may exchange *folders* when they rendezvous. Agents carry *briefcases* which are collections of *folders*. *folders* may contain state information which is unpacked at the destination and allows the agent to continue execution. Agents exchange information by *meeting* each other at rendezvous sites and exchanging folders. TACOMA implements a *rear guard* agent to deal with site failures. The TACOMA notion of *folders* is similar to the notion of *ObjectGroup* in Sumatra. Agents pay for services using electronic cash units. We have not addressed the problem of payment for services in Sumatra.

Agent TCL [24] is a TCL based agent system which provides state mobility. Communication between agents in Agent TCL happens through explicit message passing (*send* and *receive*). The system supports security by public key encryption methods.

In addition to these systems that support mobility, a number of distributed programming systems have been developed to ease the task of developing communicating programs. Bal, Stienter and Tanenbaum give a survey of these systems [11].

While the design of Sumatra was influenced by several

other systems such as Emerald, Obliq and Telescript, Sumatra differs from these systems in a few respects. The most significant difference is that data movement is under application control. There is no automatic motion of objects. For example when an application accesses public members of a class that is not local, an *IllegalAccess* exception is generated. The application may then react by moving the execution to the Engine where the remote object resides or move the remote object to the location where the application is running or react in some other "application specific" manner <sup>6</sup>. We believe that a high degree of application control in deciding about migration and data movement is key in network aware mobile applications.

The focus of our work has been to make distributed applications achieve better performance using mobility as a tool to adapt to resource variations. We have therefore not as yet addressed the important issues of security and resource usage containment our implementation.

Application-transparent or system level adaptation to wide variations in network bandwidth has been used successfully by the designers of the CODA file system [18] to improve the performance of applications.

Several experimental studies have been conducted on various aspects of Internet performance.

Sanghi et al [5] present studies on round trip time measurements of UDP packets. Their observations show that round trip times show significant variability with sharp peaks. If the peaks are ignored, the round trip time shows a step change behavior. This is consistent with our observations.

Paxson [21, 22] has experimentally studied the various aspects of Internet performance. His study on routing concludes that in the Internet, the majority of routes persist for days. Further, approximately half of the connections displayed routing asymmetry. Thus the expected delay going from host A to host B is likely to be different from the expected delay going from host B to host A. Paxson's study concludes that traffic various quantities of interest such as packet inter arrival time may be explained with a self similar model.

Golding [9] has investigated methods to predict RTT measurements and bandwidth using moving averages. Their work implies that bandwidth estimation using previous observations alone as a basis for prediction does not work well.

Carter and Crovella [19] propose a bandwidth probing scheme to estimate the bandwidth of a connection. They observe that congestion is usually caused by queuing delays at a bottleneck link. The base bandwidth of the connection is estimated by the speed of the bottleneck link. This is done by sending packets in succession such that there is

---

<sup>6</sup>The exception handling facilities of Java come in handy here.

a high probability that they will queue one after the other at the bottleneck router. The packet inter-arrival time is a measure of how fast the bottleneck router can process the packets and hence indicates the base bandwidth. A decision filtering method is used to eliminate false measurements caused by various effects such as downstream congestion, packet loss, competing traffic and queuing failure.

## 8 Conclusions and future work

This paper is a first step in demonstrating that distributed programs can use mobility as a tool to adapt to variations in their operating environment. Our exploration of network-aware mobile programs lead us to the following conclusions. First, network-aware placement of components of a distributed application can provide significant performance gains over a network-oblivious placement. For short term applications (applications that run for an hour or so) exploiting spatial variations as well as variations in the number and location of the clients achieves most of the gains. For longer-running applications, exploiting temporal variations might be worthwhile. Second, effective mobility decisions can be based on coarse-grained monitoring. This allows cheap active monitoring without losing effectiveness. Finally, there is significant spatial and temporal variation in Internet latency which can be effectively adapted to by mobile objects.

We believe that there is a class of long running applications over the Internet for which resource-aware mobility could provide flexibility and performance which would take a lot more effort to achieve by other means. One future direction we would like to pursue is to identify such applications and understand their structure and requirements. Some of the examples we intend to study include resource-aware pre-fetching for web clients, sequence servers and multi-database queries over the Internet. Another direction that we plan to explore is efficient distributed monitoring of other resources, in particular, network bandwidth and server availability. We are investigating cheap methods of estimating network bandwidth. An important question that we are investigating is how accurate resource estimates need to be in order to benefit from resource aware mobility and how the accuracy of estimation affects performance.

## Acknowledgements

The authors are grateful to Mustafa Uysal, Manuel Ujaldon and the anonymous referees for their suggestions.

## References

- [1] Java object serialization specification. <http://chatsubo.javasoft.com/current/serial/index.html>.
- [2] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *ACM SIGOPS*, pages 217–229, Dec. 1993.
- [3] L. Cardelli. A language with distributed scope. In *22<sup>nd</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1995. <http://www.research.digital.com/SRC/Obliq/Obliq.html>.
- [4] C. ching Chang. Rexdc - a remote execution mechanism. In *ACM SIGCOMM*, pages 106–115, 1989.
- [5] D.Sanghi, A.K.Agrawala, O.Gudmundsson, and B.N.Jain. Experimental assessment of end-to end behavior on internet. Technical Report CS-TR-2909, University of Maryland, June 1992.
- [6] E.Jul, H.Levy, N.Hutchinson, and A.Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(2):109–133, Feb. 1988.
- [7] E.Zayas. Attacking the process migration bottleneck. In *11th. ACM Symposium on Operating Systems Principles*, pages 13–24, Nov. 1987.
- [8] F.Douglis and J.Ousterhout. Process migration in the Sprite operating system. In *7<sup>th</sup> International Conference on Distributed Computing Systems*, pages 18–25, Sept. 1987.
- [9] R. A. Golding. End-to-end performance prediction for the internet (work in progress). Technical Report UCSC-CRL-92-26, University of California at Santa Cruz, June 1992.
- [10] J. Gosling and H. McGilton. The Java language environment white paper, 1995. Available at : <http://www.java.sun.com>.
- [11] H.E.Bal, J.G.Stiener, and A.S.Tanenbaum. Programming languages for distributed computing. *ACM Computing Surveys*, 21(3), 1989.
- [12] G. M. Inc. Telescript Technology: Mobile Agents. Available at : <http://www.genmagic.com/Telescript/Whitepapers>.
- [13] G. M. Inc. Telescript Language Language Reference and Users Guide, 1995. Available at : <http://www.genmagic.com/Telescript/TDE>.
- [14] D. Johansen, R. van Renesse, and F. B. Schneider. An introduction to the tacoma distributed system version 1.0. Technical Report 95-23, University of Tromso, 1995. Available at : <http://tklab1.cs.uit.no/DOS/Tacoma/Publications.html>.
- [15] J.R.Falcone. A programmable interface language for heterogeneous systems. *ACM Transactions on Computer Systems*, 5(4):330–351, Nov. 1987.
- [16] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, Texas, 1987.
- [17] J.W.Stamos and D.K.Glifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

- [18] L.B.Mummert, M.R.Ebling, and M.Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th. A.C.M Symposium on Operating Systems Principles*, Dec. 1995.
- [19] R. L.Carter and M. E.Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Boston University, 1996.
- [20] M. Litzkow and M. Livny. Experiences with the condor distributed batch system. In *IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL., 1990. Available at : <http://www.cs.wisc.edu/condor/publications.html>.
- [21] V. Paxson. End-to-end routing behavior in the internet. In *ACM SIGCOMM*, Aug. 1996. Available at : <http://ftp.ee.lbl.gov/nrg-papers.html>.
- [22] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995. Available at : <http://ftp.ee.lbl.gov/nrg-papers.html>.
- [23] M. Ranganathan, A. Acharya, and J. Saltz. Distributed resource monitors for mobile objects. In *International Workshop on Operating System Support for Object Oriented Systems*, 1996. Available at : <http://www.cs.umd.edu/acha>.
- [24] R.S.Gray. Agent tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996. Available at : <http://www.cs.dartmouth.edu/agent/papers.html>.
- [25] S.M.Clamen, L.D.Lebengood, S.M.Netlles, and J.M.Wing. Reliable distributed computing with avalon/common lisp. In *International Conference on Computer Languages*, pages 169–179, New Orleans, LA, 1990.
- [26] J. M. Smith. A survey of process migration mechanisms. In *Operating Systems Review*, May 1988.